Einführung in die moderne
Assembler-Programmierung
RISC-V spielerisch und fundiert lernen

DIE LESEPROBE

Thema

Ein Buch wie dieses steht vor der Herausforderung, unterschiedlichen Lesergruppen gerecht zu werden: von Neulingen an der Universität, die vielleicht zum ersten Mal in die Eingeweide eines Prozessors blicken, bis hin zu Vollprofis, die vermutlich bereits mehr über Assembler vergessen haben, als der Autor jemals wissen wird. Wir werden für die erste Gruppe am Anfang Begriffe erklären und Hintergründe erläutern. Je tiefer wir in die Materie eindringen, desto mehr Wissen setzen wir als bekannt voraus.

Was ist Assembler (ganz kurz)?

Eigentlich sind Computer fürchterliche Geräte. Sie verstehen nur Zahlen, was Menschen langfristig nicht guttut. Zudem gehen sie in winzigen Arbeitsschritten vor, was uns nur deswegen nicht auffällt, weil sie jeden einzelnen dieser Schritte *sehr* schnell ausführen. Auf dieser untersten, menschenfeindlichen Ebene der Programmierung sprechen wir von der »Maschinensprache«, auf Englisch *machine language*.

Um die Arbeit mit Computern für Nicht-Freaks erträglich zu machen, wird die Maschinensprache unter einem Berg von Abstraktionen versteckt. Dazu gehören Hochsprachen wie Python oder Rust, die von speziellen Programmen wie Compilern oder Interpretern über Zwischenschritte in Maschinensprache übersetzt werden. Bei einer Low-Code-Entwicklungsumgebung werden grafische Elemente zusammengeklebt, sodass – wenn überhaupt – nur »niedrige« Programmierkenntnisse erforderlich sind. Die gegenwärtig höchste Abstraktionsstufe sind die Anweisungen an eine Künstliche Intelligenz (KI), die »Prompts«.

»Assembler« ist in diesem Modell die erste Abstraktionsebene nach der Maschinensprache. Das Wort kommt von dem englischen Verb *to assemble,* hier im Sinne von »aus Einzelteilen zusammensetzen« und nicht von »Menschen sammeln sich« wie der Schlachtruf *Avengers assemble!* bei Marvel. Vereinfacht gesagt wird dabei jeder Maschinenbefehl 1:1 durch einen Assembler-Befehl abgebildet, der aus mehr oder weniger leicht zu merkenden Abkürzungen besteht. Das

Problem der winzigen Arbeitsschritte bleibt. Der Katalog aller Befehle eines Prozessors ist sein »Befehlssatz«, auf Englisch *instruction set*.

Da jede Prozessorfamilie eine andere Maschinensprache mitbringt, die sich zum Teil auch noch von Prozessormodell zu Prozessormodell unterscheidet, gibt es viele Varianten von Assembler. Wir sprechen von dem jeweiligen Befehlssatz eines Modells oder einer Chip-Familie, der *instruction set architecture* (ISA). Neben dem RISC-V-Befehlssatz gibt es etwa den x86 bei Intel und AMD bei klassischen PCs sowie die Arm-Befehlssätze insbesondere bei mobilen Geräten.

Der unvermeidliche geschichtliche Überblick

> *Dieses Programm hätte die traumhafte Laufzeit von 32 Millisekunden, aber den alptraumhaften Speicherbedarf von 24 KByte.*
>
> *–Peter-Mattias Oden, Grafik-Tuning. Schneller Bildaufbau mit 6502-Prozessoren am Beispiel des Apple II (1984)*

Als Erfinderin von Assembler gilt Kathleen Booth vom Birkbeck College in London – das war 1947. [BK] Zwar wurde sogar noch die erste Version des IBM-Mainframe-Betriebssystems OS/360 1966 in Assembler geschrieben. Allerdings ging es dann wegen des Aufstiegs von höheren Sprachen wie Fortran, Lisp und Cobol bergab. Assembler galt als tot. [SD]

Der Aufstieg der Acht-Bit-Computer auf der Grundlage von Prozessoren wie dem 6502 oder Z80 Mitte der 70er Jahre brachte eine Auferstehung. [SD] Zunächst gab es keine Compiler für diese Mikroprozessoren (MPU). Wer die maximale Leistung aus der Kiste holen wollte, kam an Assembler nicht vorbei. Computerzeitschriften wie *c't* waren in den 80er Jahren entsprechend noch ein Stück mehr *hardcore* als heute und muteten ihren Lesern seitenweise Listings von Assembler-Mathematik und -Grafikcode zu. Assembler auf diesen Prozessoren machte einfach *Spaß*, nicht zuletzt weil die Befehlssätze für Menschen geschrieben wurden.

Mit einer größeren Leistungsfähigkeit der Rechner wurden allerdings auch die Compiler immer besser. Der Aufwand, einen schnelleren Code per Hand zu schreiben, lohnte sich immer weniger. Zudem traten die x86-Prozessoren ihre jahrzehntelange Dominanz an mit riesigen, barocken Befehlssätzen, die für Normalsterbliche kaum zu durchdringen waren und dazu noch mit Nutzungseinschränkungen bewehrt sind. Assembler wurde bestenfalls für Bootloader verwendet, und wer das machen musste, tat es meistens fluchend. Die Befehlssätze werden seitdem und bis heute für Compiler entworfen. Wenn es um die Praxis ging, schien Assembler tot, schon wieder.

Also warum dann jetzt noch Assembler?

> *I could skip the middleman and talk right to the machine. (…) I could talk to God, just like IBM.* [TK]
>
> *–Tracy Kidder, The Soul of a New Machine (1981)*

Vermutlich sollte an dieser Stelle eine flammende Rede für den Nutzen von Grundwissen über Assembler in der Computerwissenschaft stehen – dass erst damit klar wird, wie die Maschine auf der untersten Ebene funktioniert, dass dieses Wissen für den Compilerbau unabdingbar ist, dass es bei der Optimierung von Code helfen kann. Das ist auch alles wahr.

Allerdings müssen wir der Realität ins Auge sehen: Eine Menge Leute lernen im 21. Jahrhundert Assembler nur, weil es Teil eines Pflichtkurses an der Uni ist. Deswegen ist dieses Buch erstens kurz und zweitens befasst es sich mit RISC-V. Wie wir gleich ausführlicher sehen werden, geht dieser Befehlssatz auf frustrierte Informatik-Dozenten zurück, die unter anderem ein besseres Werkzeug für die Lehre haben wollten. Der minimalistische Kern-Befehlssatz kann selbst den desinteressierten Massen im Grundkurs zugemutet werden, die danach nie wieder irgendwas mit Assembler zu tun haben (wollen).

Profis werden dagegen vermutlich die ersten Teile des Buches überspringen und sich auf RISC-V konzentrieren wollen. Es gibt immer noch Situationen, wo es Assembler sein muss, zum Beispiel die Portierung von Betriebssystemen auf neue Prozessoren. Der langsame, aber stetige Aufstieg von RISC-V in der Industrie zwingt mehr Leute dazu, sich mit diesem Neuling zu beschäftigen. Diese Teile des Buches sind daher auch zum Nachschlagen gedacht und absichtlich etwas nüchterner geschrieben (wenn auch nicht viel). Profis haben ja keine Zeit.

Langfristig sollten diese beiden Lesergruppen deckungsgleich werden: Ein Ziel von RISC-V ist, dass im Studium derselbe Befehlssatz gelehrt wird, der auch in der Praxis verwendet wird. Die Neueinstellungen würden dann nützliches, sofort einsetzbares Wissen von der Uni mitbringen, was einem kleinen Wunder gleichkäme.

Hobby-Coder als dritte Gruppe sind dagegen nicht nur die Spiel-, sondern auch die Glückskinder der Programmierwelt. Sie können tun, was sie wollen, in welcher Sprache sie es wollen, so lange, wie sie es wollen. Ihre Projekte können den praktischen Nutzen eines überfahrenen Stinktiers haben. Assembler coden sie entsprechend zum Spaß, auch wenn ihnen besorgte Bekannte T-Shirts schenken mit Schriftzügen wie »Hauptsache, es tut weh«.

Der große Feind des Hobby-Coders sind Updates und neue Features. Neben Familie und Job und was sonst noch im Leben wirklich wichtig ist bleibt ein Projekt manchmal so lange liegen, bis sich Staub auf der Tastatur sammelt. Wer sich dann erst in neue Frameworks, Funktionen oder Updates reinfuchsen muss,

kommt weniger zum Programmieren. Lizenzbedingungen können ein weiteres Problem sein. Es ist daher kein Wunder, dass sich viele Freizeit-Assembler-Fans in die Retro-Programmierung etwa des 6502 aus dem Acht-Bit-Zeitalter geflüchtet haben. RISC-V macht jetzt damit Schluss: Der Befehlssatz ist nicht nur kurz, sondern auch »eingefroren« und ändert sich nicht wieder.

Für diese Gruppe ist insbesondere der dritte und letzte Teil des Buches gedacht. Die dort vorgestellten Routinen, Verfahren und Programme würden von vernünftigen Menschen in einer Hochsprache geschrieben (oder gar nicht), der Stoff bietet jedoch tiefere Einblicke in die Assembler-Welt. Für die ganz Harten diskutieren wir schließlich am Ende noch weitergehende Projekte, die zu umfangreich für dieses Buch wären. An ihnen dürften nur noch zwei Gruppen Freude haben: sadistische Dozenten und masochistische Hobby-Coder. T-Shirts für alle!

Was ist RISC-V?

> *The most pervasive change in this edition is switching from MIPS to the RISC-V instruction set. We suspect this modern, modular, open instruction set may become a significant force in the information technology industry. It may become as important in computer architecture as Linux is for operating systems.*
>
> *–Hennessy und Patterson, Computer Architecture: A Quantitative Approach (2019)*

RISC-V wird auf Englisch *»risk five«* ausgesprochen und als »Risk fünf« eingedeutscht, auch wenn einige KIs gegenwärtig noch auf *»risk vee«* bestehen. Der erste Teil des Namens kommt von *Reduced Instruction Set Computer* und bezeichnet Prozessoren, die über vergleichsweise wenige Befehle verfügen, aber diese sehr schnell ausführen. Dabei wird etwas mehr Code benötigt als bei einem *Complex Instruction Set Computer* (CISC). Die römische Ziffer V verweist darauf, dass es der fünfte Anlauf der Erfinder ist. Das RISC-V-Projekt ist vergleichsweise jung, in der heutigen Form nahm es 2010 seinen Anfang. Über die Einhaltung der Standards wacht seit 2020 die Stiftung RISC-V International mit Sitz in der Schweiz.

Als Befehlssatzdefinition existiert RISC-V eigentlich nur auf dem Papier. Sie besteht aus einer Spezifikation, die nichts darüber aussagt, wie der Prozessor die Befehle umsetzt. Ob konventionelle Hardware wie Logikgatter, elektromagnetische Relaistechnik wie zu Zeiten von Konrad Zuse oder dressierte Hamster in speziellen Laufrädern, alles ist möglich.

Unter uns gesagt: Der Befehlssatz an sich ist nicht fürchterlich aufregend und schon gar nicht revolutionär. Wer sich bereits mit der ISA von anderen RISC-Pro-

zessoren beschäftigt hat, wird vieles wiedererkennen. Vielmehr zeichnen RISC-V zwei Dinge aus:

Erstens, der Standard ist »offen« oder »frei«, denn die Spezifikation unterliegt einer Creative-Commons-Lizenz. Damit kann jeder selbst RISC-V-Prozessoren bauen, ob als Bastelfreak im Hobbykeller, multinationaler Konzern mit eigener Chip-Fertigung oder Verein für ambitionierte Hamster-Trainer. Forschung und Lehre sind keine Grenzen gesetzt, Unternehmen müssen keine Lizenzgebühren bezahlen und Freizeit-Coder bekommen keine Auflagen aufgedrückt.

Zweitens, es handelt sich um einen »modularen« Standard. Während der x86-Befehlssatz durch sein unablässiges Wachstum inzwischen bei einer vierstelligen Zahl von Instruktionen angekommen ist, werden die RISC-V-Befehle in »Module« verpackt. [HS] Diese werden nach eingehender Prüfung »eingefroren« *(frozen)* und nie wieder verändert. Es gibt ein Basismodul I, das alle RISC-V-Prozessoren haben müssen. Darüber hinaus entscheidet jede Chip-Schmiede und jeder Hamster-Trainer selbst, welche Module sie benötigen.

Formalitäten

So weit ein erster Überblick. Leider kommt kein Buch ohne Bürokratie aus. Bringen wir sie schnell hinter uns.

Englisch

> *Deutsch im Code sagt dem Leser auf den ersten Blick: Hier hat jemand nur für sich selbst programmiert, ohne damit zu rechnen, dass sich jemals jemand anders für den Code interessieren könnte. Das tun überwiegend Anfänger, also ist der Code wahrscheinlich nicht besonders gut.*
>
> *–Passig und Jander, Weniger schlecht programmieren (2013)*

Jedes deutschsprachige Buch über Computer muss damit klarkommen, dass Englisch die Weltsprache der Informatik ist. Besonders bei RISC-V liegt bislang viel Literatur nur auf Englisch vor. Früher oder später kommt niemand daran vorbei. Der Einsatz von Künstlicher Intelligenz verstärkt diesen Effekt nur, weil die Modelle gegenwärtig deutlich besser mit Englisch zurechtkommen als mit Deutsch. *Sorry.*

Die gute Nachricht ist, dass die erforderlichen Englischkenntnisse eher auf der Sprachebene von *Friends* liegen als von William Shakespeare. Auch Englischmuffel kommen mit etwas Übung klar. Wir führen am Anfang die englischen Fachbegriffe ein und setzen sie dann nach und nach als bekannt voraus. Auch die Kommentare in den Quelltexten *(source code)* sind irgendwann durchgängig auf Englisch, weil das der Situation in der wirklichen Welt entspricht.

Code

Ein Ziel dieses Buches ist es, möglichst viele gut lesbare Code-Beispiele zur Verfügung zu stellen. Dabei steht insbesondere am Anfang die Klarheit des Designs im Vordergrund. Tricks, um ein Maximum an Leistung oder den kürzesten Code herauszukitzeln, führen wir erst ein, wenn das Grundprinzip klar ist. Die Programme sind ausführlich kommentiert. Wer schon mal versucht hat, fremden Assembler-Code zu lesen – oder nach einigen Wochen den eigenen –, weiß, warum. Ein Kommentar pro Zeile wird keine Seltenheit sein.

Eine historisch gewachsene Unsitte bei Assembler ist die Verwendung von sehr kurzen Namen oder gar einzelnen Buchstaben für Variablen und Sprungmarken *(label)*. Dafür gibt es im 21. Jahrhundert keine Entschuldigung, wir verwenden lange Namen. Konstanten werden in VERSALIEN geschrieben, auch wenn es der Maschine egal ist.

Die Grobstruktur von Routinen wird meist so aussehen, dass wir ganz oben einsteigen und ganz unten wieder rausgehen. Anders formuliert soll jede Routine möglichst immer nur einen Eingang und einen Ausgang haben. Im Rahmen des *defensive programming* bauen wir hin und wieder Code ein, der nur dazu dient, das Programm robuster zu machen. Entsprechende Stellen markieren wir in den Kommentaren als *paranoid*. Wir sagten ja bereits, hier sind Wahnsinnige am Werk.

Werkzeuge

Die Beispielprogramme wurden entweder auf dem RARS-Simulator (*https://github.com/TheThirdOne/rars*) oder mit QEMU unter Ubuntu Linux mit dem GCC Compiler (*https://gcc.gnu.org/*) getestet. RARS ist der einfachere Weg. Das Programm wird als ausführbare `jar`-Datei bereitgestellt und müsste auf ziemlich jedem Betriebssystem mit

```
java -jar <DATEI>
```

von der Kommendozeile aus ausführbar sein.

GCC ist dafür deutlich mächtiger. Ubuntu bietet für QEMU ein vorgefertigtes Image unter *https://wiki.ubuntu.com/RISC-V* an, das ein komplettes RISC-V-System emuliert. Das heißt, wir können innerhalb dieser Umgebung mit normalen Werkzeugen arbeiten. Für dieses Buch wurde benutzt:

```
ubuntu-22.04.1-preinstalled-server-riscv64+unmatched.img
```

Wir rufen die QEMU-Instanz auf mit:

```
qemu-system-riscv64 \
-machine virt \
-nographic \
-m 2048 \
-smp 4 \
-bios /usr/lib/riscv64-linux-gnu/opensbi/generic/fw_jump.elf \
-kernel /usr/lib/u-boot/qemu-riscv64_smode/uboot.elf \
-device virtio-net-device,netdev=eth0 \
-netdev user,id=eth0,hostfwd=tcp::10022-:22 \
-drive file=<UBUNTU_IMAGE>,format=raw,if=virtio
```

Wir können uns dann von einem anderen Rechner aus mit `ssh -p 10022 ubuntu@<RECHNER>` einloggen.

That said, above all this book tries not to take itself (or anything) too seriously. There is humour here, the difference is that you need to look for it.

–Doug Hoyte, Let Over Lambda (2008)

10 Der Befehlssatz

After weighing our options, we embarked on what we expected would be a semester-long effort to make a clean-slate design of a new ISA. To say we underestimated the task would be a charitable understatement: we completed the user-level instruction set architecture four years later.

–Andrew Waterman, Design of the RISC-V Instruction Set Architecture (2016)

In diesem Buch behandeln wir die Befehle von zwei Modulen ausführlich: das Basismodul `I` (für *integer*) und `M` (für *multiplication*). Das `I`-Modul ist das Herz von RISC-V, in jedem Prozessor enthalten und besteht aus etwa 40 Einzelbefehlen. Vom `M`-Modul kommen noch acht hinzu.

10.1 Laden und speichern

Die mit Abstand wichtigsten Befehle bei RISC-V sind die zum Laden einer Zahl in ein Register, denn als *load-and-store* CPU kann der Prozessor nur Daten bearbeiten, die dort vorliegen. Unter den vielen Ladebefehlen sind wiederum die folgenden zwei die wichtigsten:

Befehl	Beschreibung	Format	Funktion	Beispiel
`li`	*Load Immediate*	li rd, i	rd ← i	li t0, 0x2C03E
`la`	*Load Address*	la rd, symbol	rd ← addr	la t0, my_string

10.1.1 Load Immediate `li`

Dieser Befehl dient dazu, eine explizit angegebene Zahl in ein Register zu laden. Allerdings gehört es zum guten Stil, nicht direkt Zahlen zu nehmen: In der Form wie oben liegen sie sonst im Code verstreut als *magic numbers*, die wir uns bei etwaigen Änderungen mühsam zusammensuchen müssen.

```
.eqv JONES 0x2C03E
        ...
        li t0, JONES
```

Die einzig wahre Null

Es gibt bei RISC-V viele Möglichkeiten, um ein Register zu »löschen«, also dort eine Null zu laden. Wir nehmen immer `li rd, 0`, weil das unserer Absicht entspricht. Insbesondere verzichten wir auf irgendwelche neunmalklugen Tricks wie `xor t0, t0, t0`, die bei anderen Prozessoren gängig sind.

10.1.2 Load Address `la`

Der beste Freund von `li` heißt *Load Address* (`la`). Bei der Programmierung von modernen Prozessoren benutzen wir fast nie absolute Adressen, sondern Label. Der Assembler rechnet das in Zusammenarbeit mit dem Linker für uns automatisch um. Mit `la` lassen wir diese Adressen erzeugen und laden sie in ein Register.

```
alices_cat_axiom:
        .asciz "A cat may look at a king"
                ...
                la t0, alices_cat_axiom
```

Als Variante von `la` gibt es den Befehl `lla` für *Long Load Address*, der eine Adresse aus einer weit entfernten Speicherstelle lädt. Verwendet wird der Befehl genauso wie `la`.

Der Unterschied zwischen `la` und `li` ist am Anfang nicht immer klar – warum können wir nicht einfach das Label `msg` auch mit `li` in ein Register laden? Das Problem besteht darin, dass für die endgültige Adresse auch der aktuelle Stand des Programmzählers `pc` benötigt wird. Auf die Details gehen wir später ein. Für den Moment merken wir uns: `li` für Zahlen und `la` für Adressen. Einige Assembler unterstützen uns dabei, indem sie negative Werte für die Operanden von `la` nicht zulassen.

AI-Probleme mit A und I

Der Unterschied ist nicht nur für Menschen am Anfang etwas verwirrend. Bei KI-generiertem Code finden wir vergleichsweise häufig eine Verwendung von `li` statt `la` – zumindest gegenwärtig noch.

10.1.3 Move `mv`

Wenn wir erstmal eine Zahl in einem Register haben, können wir sie von dort aus mit *Move* (`mv`) in ein anderes kopieren.

Befehl	Beschreibung	Format	Funktion	Beispiel
`mv`	*Move*	mv rd, rs	rd ← rs	mv t1, t0

Nach diesem Befehl steht in `rd` der gleiche Wert wie in `rs`. Der Vorgang ist damit eigentlich kein *move*, sondern ein *copy*, aber der Begriff hat sich eingebürgert. Allgemein gilt: RISC-V-Befehle sind nicht destruktiv, was heißt, dass der ursprüngliche Wert in einem Register nicht gelöscht wird.

Ein Register kann nicht *indirekt* auf ein anderes zugreifen. Es gibt damit keine Möglichkeit, eine Schleife über die Inhalte von `t0` bis `t3` auszuführen. Um die Register `t0` bis `t3` zu löschen, brauchen wir vier einzelne Befehle.

10.1.4 Laden aus dem Speicher

Die Befehle dieser Gruppe beginnen mit dem Buchstaben »l« wie *load*, gefolgt von einer Abkürzung für die Wortbreite, also ein Buchstabe aus der Liste »bhwdq«. In ihrer einfachsten Form greifen sie auf eine Adresse im Speicher zu:

Befehl	Beschreibung	Format	Funktion	Beispiel
`lb`	*Load Byte*	lb rd, symb	rd ← M(symb)[7:0]	lb t0, jennings
`lh`	*Load Halfword*	lh rd, symb	rd ← M(symb)[15:0]	lh t0, wescoff
`lw`	*Load Word*	lw rd, symb	rd ← M(symb)[31:0]	lw t0, lichterman
`ld`	*Load Doubleword*	ld rd, symb	rd ← M(symb)[63:0]	ld t0, snyder
`lq`	*Load Quadword*	lq rd, symb	rd ← M(symb)[127:0]	lq t0, bilas

Dabei kommt `ld` bei RV64 hinzu und `lq` bei RV128. Ein Beispiel macht es klarer:

```
                lb t0, mcnulty
                ...
mcnulty:
        .byte 0xE6
```

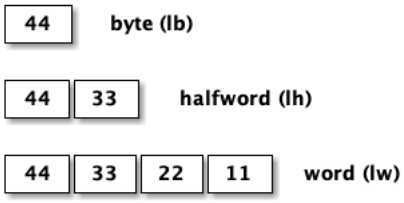
Wie häufig bei RISC-V können wir aber auch eine andere Variante nutzen, die auf ein Register zugreift, das die eigentliche Adresse enthält.

Befehl	Beschreibung	Format	Funktion	Beispiel
`lb`	*Load Byte*	lb rd, i(rs)	rd ← M(rs+i)[7:0]	lb t0, 0(t1)
`lh`	*Load Halfword*	lh rd, i(rs)	rd ← M(rs+i)[15:0]	lh t0, 1(t1)
`lw`	*Load Word*	lw rd, i(rs)	rd ← M(rs+i)[31:0]	lw t0, 2(t1)
`ld`	*Load Doubleword*	ld rd, i(rs)	rd ← M(rs+i)[63:0]	ld t0, -1(t1)
`lq`	*Load Quadword*	lq rd, i(rs)	rd ← M(rs+i)[127:0]	lq t0, -2(t1)

```
my_number:
        .word 0x11223344
                ...
                la t0, my_number

                lb t1, 0(t0)
                lh t2, 0(t0)
                lw t3, 0(t0)
```

Am Ende enthält `t1` das Byte 0x44, `t2` das Halbwort 0x3344 und `t3` das Wort 0x1122_3344.



Der Offset »i« ist maximal 12 Bit breit und kann damit Zahlen von 0x800 (-2048) bis 0x7FF (2047) darstellen – ein Adressraum von 4 KiB. Bei diesen Befehlen wird das Vorzeichen erweitert.

10.1.5 Kampf der Vorzeichenerweiterung

Nicht immer wollen wir eine Vorzeichenerweiterung, etwa wenn wir ein Byte von einer Schnittstelle laden. Um sie auszuschalten, können wir die Zahl mit den Befehlen `lbu` und `lhu` – mit einem »u« wie *unsigned* – ohne Vorzeichen in ein Register laden. Bei `lbu` ist `0xFF` dann 255 und nicht -1. Wir geben hier nur die Varianten mit einem zweiten Register an:

Befehl	Beschreibung	Format	Funktion	Beispiel
`lbu`	*Load Byte Unsigned*	lbu rd, i(rs)	rd ← M(rs+i)[7:0]	lbu t0, 0(t1)
`lhu`	*Load Halfword Unsigned*	lhu rd, i(rs)	rd ← M(rs+i)[15:0]	lhu t0, t1(a0)

Für ASCII benutzen wir `lbu`, für 16-Bit-Unicode `lhu`. Bei RV32 gibt es kein `lwu`, weil das Register durch den 32-Bit-Wert komplett ausgefüllt wird. Wir finden diesen Befehl dagegen bei RV64.

10.1.6 Speicherbefehle

Die Speicherbefehle entsprechen im Wesentlichen den Ladebefehlen. Sie fangen mit »s« für *store* an.

Befehl	Beschreibung	Format	Funktion	Beispiel
sb	*Store Byte*	sb rs, symb, rt	rs[7:0] → mem(symb)	sb t0, billing, t1
sh	*Store Halfword*	sh rs, symb, rt	rs[15:0] → mem(symb)	sh t0, piloty, t1
sw	*Store Word*	sw rs, symb, rt	rs[31:0] → mem(symb)	sw t0, lehmann, t1
sd	*Store Doubleword*	sd rs, symb, rt	rs[63:0] → mem(symb)	sd t0, petri, t1
sq	*Store Quadword*	sq rs, symb, rt	rs[127:0] → mem(symb)	sq t0, walk, t1

```
zuse:
        .space 8
                ...
                li t0, 0x1234
                sw t0, zuse, t1
```

Dinah ist aus dem Sack

Es ist auf den ersten Blick nicht klar, was das zweite Register `rt` da eigentlich soll – reicht es nicht zu sagen, dass wir den Inhalt des Quellregisters `rs` an der Speicherstelle `symb` abspeichern wollen? Hier lässt sich die furchtbare Wahrheit nicht länger verbergen: Die Speicherbefehle mit Symbolen gibt es eigentlich gar nicht, sie werden uns nur vorgegaukelt durch den Assembler als »Pseudo-Befehle«. Unter der Motorhaube muss ein Register als Zwischenspeicher benutzt werden. Wir erklären die hässlichen Einzelheiten später.

Auch hier gibt es Varianten mit Registern.

Befehl	Beschreibung	Format	Funktion	Beispiel
sb	*Store Byte*	sb rs, i(rd1)	rs → mem(rd1+i)	sb t0, 8(t1)
sh	*Store Halfword*	sh rs, i(rd1)	rs → mem(rd1+i)	sh t0, 8(t1)
sw	*Store Word*	sw rs, i(rd1)	rs → mem(rd1+i)	sw t0, 8(t1)
sd	*Store Doubleword*	sd rs, i(rd1)	rs → mem(rd1+i)	sd t0, 8(t1)
sq	*Store Quadword*	sq rs, i(rd1)	rs → mem(rd1+i)	sq t0, 8(t1)

Der Offset ist am Anfang etwas gewöhnungsbedürftig, aber damit können wir uns unter anderem Schleifen ersparen. Nehmen wir an, wir wollen bei einer RV32-Maschine die 16 Byte ab einer Speicherstelle mit 0xFF vollschreiben.

```
li a0, 0xFFFFFFFF        # a word of bit
la t0, van_der_mey

sw a0, 0(t0)             # Clear first four byte
sw a0, 4(t0)             # ... second
sw a0, 8(t0)             # ... third
sw a0, 12(t0)            # ... fourth
```

```
                ...
van_der_mey:
                .space 16
```

10.2 Mathe und Logik

Genug geladen, jetzt wird gearbeitet. Wir fangen mit den Befehlen des Rechenwerks (*arithmetic logic unit*, ALU) an.

10.2.1 Addition und Subtraktion

»Can you do Addition?« the White Queen asked. »What is one and one and one and one and one and one and one and one and one and one?« »I don't know,« said Alice. »I lost count.« »She ca'n't do Addition,« the Red Queen interrupted. »Can you do Subtraction?«

–Lewis Carroll, Through the Looking-Glass and What Alice Found There (1871)

Addition und Subtraktion kommen im gewohnten Format mit drei Operanden daher.

Befehl	Beschreibung	Format	Funktion	Beispiel
`add`	*Add*	add rd, rs1, rs2	rd ← rs1 + rs2	add t0, t0, t1
`addi`	*Add Immediate*	addi rd, rs1, i	rd ← rs1 + i	addi t0, t0, -4
`sub`	*Subtract*	sub rd, rs1, rs2	rd ← rs1 - rs2	sub t0, t0, t1

Bei der Immediate-Variante kann die angegebene Zahl auch hier höchstens 12 Bit lang sein, selbst bei RV64. Es gibt keinen Befehl `subi`, denn das ist nur eine Addition mit negativem Vorzeichen.

Der subi-Sonderfall

Wegen der Asymmetrie des Zweierkomplements gibt es einen Fall, bei dem ein `subi` nicht durch ein `addi` mit einem negativen Wert ersetzt werden kann: Eine `subi`-Subtraktion mit einem Subtrahenden von -2^{11} würde 2^{11} zu dem Zielregister hinzuaddieren. Das geht mit `addi` nicht. [WA]

Wer Erfahrung mit anderen Prozessoren hat, wird jetzt Begriffe wie *carry* oder *borrow* vermissen. Tatsächlich gibt es bei RISC-V keinen eingebauten Mechanis-

mus, um einen Übertrag anzuzeigen oder gar in die Addition einzubeziehen. Er fällt einfach unter den Tisch. Das ist einer der häufigsten Kritikpunkte an RISC-V. Wir werden das später in den Griff bekommen.

Das bringt uns zu der Frage, wie wir das Vorzeichen einer Zahl ändern. Dazu nutzen wir den Befehl *Negate* (`neg`).

Befehl	Beschreibung	Format	Funktion	Beispiel
neg	*Negate*	neg rd, rs	rd ← -rs	neg t0, t0

Zurück zur Addition. Viele Assembler sind gutmütig genug, dass sie das Richtige tun, wenn wir ihnen bei einem Immediate-Wert statt `addi` den normalen Additionsbefehl `add` mit einer Zahl unterschieben:

```
add t0, t0, 1        # don't do this
```

Das schreit nach Tippfehler – aber welcher? Fehlt hier das `i` von `addi` im Opcode oder ein Buchstabe wie `t` beim letzten Operanden? Niemand weiß es. Daher immer `addi`.

Im Gegensatz zu anderen Prozessoren gibt es bei RISC-V keine gesonderten Befehle wie `inc` oder `dec`, um den Inhalt eines Registers um eins zu erhöhen oder zu verkleinern, zwei sehr häufige Aufgaben bei Schleifen. Dazu werden normale Additionsbefehle verwendet:

```
addi t0, t0, 1      # "inc t0"
addi t0, t0, -1     # "dec t0"
```

Alle lieben `addi`

`addi` ist mit Abstand der am häufigsten auftretende Befehl in RISC-V-Code. In einigen Programmen macht er knapp ein Viertel aller Instruktionen aus. [WA] Wie wir später sehen werden, liegt das insbesondere an seiner Verwendung bei Pseudo-Befehlen.

Beim RV64 gibt es Varianten dieser Befehle, die sich nur auf die unteren 32 Bit beziehen: `addiw`, `addw`, `negw` und `subw`. Dabei wird die Berechnung normal vorgenommen, aber dann werden die oberen 32 Bit verworfen. Das kann zu unerwarteten Effekten führen. Schauen wir uns die Rechnung 0x0000_0000_FFFF_FFFF (dezimal 4.294.967.295) + 2 an:

```
li t0, 0x00000000FFFFFFFF
addi t1, t0, 2                  # no problem
addiw t2, t0, 2                 # wrong
```

Eigentlich erwarten wir 0x0000_0001_0000_0001, was hier nach dem normalen `addi` auch das Ergebnis in `t1` ist. Dagegen werden bei `addiw` die oberen 32 Bit verworfen, was uns in `t2` die Zahl 0x0000_0000_0000_0001 beschert – also 1.

10.2.2 Multiplikation und Division

Multiplikation und Division sind nicht im Basismodul enthalten, dazu wird das Modul M benötigt. Im Gegensatz zu anderen Prozessoren ist die Multiplikation bei RISC-V nicht auf spezielle Register beschränkt.

Herrschen, ohne zu teilen

Neben dem Modul M gibt es das Modul Zmmul, das nur die Befehle für die Multiplikation beinhaltet. Das macht den Aufbau von kleineren Systemen einfacher, die keine Division brauchen. Denn für Assembler gilt, wie wir später genauer erläutern werden: Division ist doof.

Für die Multiplikation gibt es in M vier Befehle:

Befehl	Beschreibung	Format	Funktion	Beispiel
`mul`	*Multiply Lower*	mul rd, rs1, rs2	rd ← rs1 * rs2	mul t0, t1, t2
`mulh`	*Multiply High Signed*	mulh rd, rs1, rs2	rd ← rs1 * rs2	mulh t0, t1, t2
`mulhu`	*Multiply High Unsigned*	mulhu rd, rs1, rs2	rd ← rs1 * rs2	mulhu t0, t1, t2
`mulhsu`	*Multiply High Signed-Unsigned*	mulhsu rd, rs1, rs2	rd ← rs1 * rs2	mulhsu t0, t1, t2

Was es leider nicht gibt, ist ein Befehl nach dem Muster `muli` für eine Multiplikation mit kleinen Immediate-Zahlen als Gegenstück zu `addi`.

Wir hatten gesehen, dass RISC-V bei der Addition den Overflow ignoriert: Wer das haben will, muss gefälligst selbst dafür sorgen. Dieses »mir doch egal« zieht bei der Multiplikation nicht mehr, denn wenn wir zwei 32-Bit-Zahlen multiplizieren, ist das Ergebnis 64 Bit lang. Wir müssen daher zwischen einem »oberen« (*upper*) und einem »unteren« (*lower*) Teil des Ergebnisses unterscheiden.

Ein H für ein U vormachen

Das »h« in den Namen der Befehle steht für *high* und bezieht sich auf die oberen 32 Bit des Ergebnisses. In den Beschreibungen wird auf Englisch allerdings von *upper* gesprochen. Ein »u« für *upper* hätte allerdings mit dem bereits verwendeten »u« für *unsigned* verwechselt werden können.

Schauen wir uns das im Detail an. Der erste und einfachste Befehl lautet *Multiply Lower* (`mul`). Er gibt die unteren 32 Bit des 64 Bit langen Ergebnisses zurück.

```
mul rdl, rs1, rs2
```

Der obere Teil des Ergebnisses verschwindet einfach. Um ihn zu erhalten, benutzen wir in einem zweiten Schritt die Befehle mit dem »h« im Namen. Wir unterscheiden drei verschiedene Fälle: Multiplikand und Multiplikator mit Vorzeichen (`mulh`), ohne Vorzeichen (`mulhu`) und gemischt (`mulhsu`).

Multiplikator	Multiplikand	Befehl
mit Vorzeichen	*mit* Vorzeichen	`mulh`
ohne Vorzeichen	ohne Vorzeichen	`mulhu`
mit Vorzeichen	ohne Vorzeichen	`mulhsu`

Die Multiplikation ist damit bei RISC-V ein aus zwei Einzelbefehlen bestehender Vorgang. Sie sollen der Spezifikation zufolge möglichst in der gleichen, vorbestimmten Reihenfolge ablaufen: Erst die hohen Bit, dann die niedrigen.

```
mulh rdh, rs1, rs2
mul rdl, rs1, rs2
```

Damit können auf gewissen Systemen diese beiden Befehle zu einem verschmolzen werden durch *macro-op fusion*. Weitere Vorgabe: Das »obere« Ziel-Register `rdh` darf nicht identisch sein mit einem der Quell-Register.

10.2.3 Division

Auch für die Division haben wir vier Befehle im Modul M.

Befehl	Beschreibung	Format	Funktion	Beispiel
`div`	*Division, Signed*	div rd, rs1, rs2	rd ← rs1 / rs2	div t0, t1, t2
`divu`	*Division, Unsigned*	divu rd, rs1, rs2	rd ← rs1 / rs2	divu t0, t1, t2
`rem`	*Remainder, Signed*	rem rd, rs1, rs2	rd ← rs1 % rs2	rem t0, t1, t2
`remu`	*Remainder, Unsigned*	remu rd, rs1, rs2	rd ← rs1 % rs2	remu t0, t1, t2

Die Versionen mit einem »u« sind für *unsigned* Zahlen; `div` gibt uns das Ergebnis der Division mit erweitertem Vorzeichen, `rem` gibt uns den Rest (Modulo) und erweitert dabei auch das Vorzeichen. Auch hier gibt es eine empfohlene Vorgehensweise, wenn wir das Ergebnis und den Rest haben wollen:

```
div rdq, rs1, rs2
rem rdr, rs1, rs2
```

In der englischen Literatur werden `rdq` für *quotient* und `rdr` für *remainder* als Symbole bei der Darstellung der Division benutzt. Auch hier gilt, dass `rdq` nicht identisch sein darf mit `rs1` oder `rs2`.

Eine Besonderheit bei RISC-V besteht darin, dass kein Fehler gemeldet wird, wenn eine Division durch Null vorliegt. Um solche Fälle abzufangen, wird in der Spezifikation empfohlen, den Divisor *nach* der Rechenoperation zu prüfen – danach deswegen, weil die CPU-Einheit für die Vorhersage von Verzweigungen (*branch prediction*) dann besser funktioniert.

```
li t0, 0xEDF9C                  # dividend
li t1, 0                        # divisor, not going to work

div t2, t0, t1
rem t3, t0, t1

beqz t1, div_by_zero            # test for division by zero
```

RISC-V lässt die Maschine bei einer Division durch Null nicht in einem undefinierten Zustand. Im Quotienten werden alle Bit gesetzt und als Rest wird der Dividend abgelegt. Im obigen Beispiel erhalten wir bei RV64 in `t2` 0xFFFF_FFFF_FFFF_FFFF und in `t3` 0x0000_0000_000E_DF9C.

10.2.4 Das Modul M bei RV64

Bei RV64 wird das Modul M um fünf Befehle erweitert.

Befehl	Beschreibung	Format	Funktion	Beispiel
`mulw`	*Multiply, Lower Bits*	mulw rd, rs1, rs2	rd ← rs1 * rs2	mulw t0, t1, t2
`divw`	*Division, Signed*	divw rd, rs1, rs2	rd ← rs1 / rs2	divw t0, t1, t2
`divuw`	*Division, Unsigned*	divuw rd, rs1, rs2	rd ← rs1 / rs2	divuw t0, t1, t2
`remw`	*Remainder, Signed*	remw rd, rs1, rs2	rd ← rs1 % rs2	remw t0, t1, t2
`remuw`	*Remainder, Unsigned*	remuw rd, rs1, rs2	rd ← rs1 % rs2	remuw t0, t1, t2

Diesen Befehlen ist gemeinsam, dass sie auf 32-Bit-Daten in den 64-Bit-Registern einwirken und ein 32-Bit-Produkte erzeugen mit dem richtigen Vorzeichen. Ein `mulhw` gibt es nicht. Bei RV64 können wir ohnehin zwei 32-Bit-Zahlen mit `mul` multiplizieren und erhalten die oberen 32 Bit.

Wissen wir nicht, ob das Vorzeichen der beiden ursprünglichen 32-Bit-Zahlen richtig erweitert wurde, können wir sie beide zunächst um 32 Bit nach links verschieben und dann `mulh` oder eine der entsprechenden Varianten für *upper* verwenden. In diesem Zusammenhang werden wir den Befehl `sext.w` kennenlernen.

10.2.5 Logikbefehle

Die Opcodes tragen offensichtliche Namen und haben das übliche Dreierformat:

Befehl	Beschreibung	Format	Funktion	Beispiel
`and`	*And*	and rd, rs1, rs2	rd ← rs1 & rs2	and t0, t0, t1
`andi`	*And Immediate*	andi rd, rs1, i	rd ← rs1 & i	andi t0, t0, 1
`or`	*Or*	or rd, rs1, rs2	rd ← rs1 | rs2	or t0, t0, t1
`ori`	*Or Immediate*	ori rd, rs1, i	rd ← rs1 | i	ori t0, t0, 1
`xor`	*Xor*	xor rd, rs1, rs2	rd ← rs1 ^ rs2	xor t0, t0, t1
`xori`	*Xor Immediate*	xori rd, rs1, i	rd ← rs1 ^ i	xori t0, t0, 1

Die Immediate-Werte sind 12 Bit breit. Wie immer gibt es eine Vorzeichenerweiterung. Im Gegensatz zu den Rechenbefehlen gibt es keine `w`-Varianten.

Logikbefehle sind in der Assembler–Programmierung erstaunlich nützlich. Es kommt häufig vor, dass wir mit ihnen lange Vergleichsketten vermeiden können. Wir können zudem dafür sorgen, dass einzelne Bit in einem Register den gewünschten Wert annehmen, ohne dass die anderen Bit verändert werden.

Funktion	Befehl	Maske	Beispiel mit Ziel-Bit 0
Bit löschen	`and`	0	0xFFFF_FFFE
Bit setzen	`or`	1	0x0000_0001
Bit flippen	`xor`	1	0x0000_0001

Mit dem Befehl `ori t0, t0, 1` sorgen wir dafür, dass Bit 0 in `t0` auf jeden Fall gesetzt ist. Die anderen Bit in dem Register bleiben unberührt.

In diese Gruppe von Befehlen gehört das Einerkomplement `not`, bei dem die Bit umgekehrt werden:

Befehl	Beschreibung	Format	Funktion	Beispiel
`not`	*Not*	not rd, rs	rd ← ~rs	not t0, t1

So wird aus 0x0000_FFFF dann 0x_FFFF_0000.

Da waren's nur noch zwei

Haben wir nicht gesagt, dass RISC-V-Befehle drei Operanden haben? Das gilt auch weiterhin. Allerdings ist `not` wieder ein Pseudo-Befehl, der intern zu `xori rd, rs, -1` übersetzt wird.

Pflicht in jeder Diskussion über Assembler ist der Trick, zwei Register mit `xor` zu tauschen, ohne ein drittes zu benutzen.

```
beq t0, t1, done     # skip if already equal
xor t0, t0, t1
xor t1, t0, t1
xor t0, t0, t1
done:
```

Bei 32 Registern ist das allerdings meist neunmalkluger Schnickschnack.

10.2.6 Schiebebefehle

Das ist die letzte Gruppe der ALU-Befehle.

Befehl	Beschreibung	Format	Funktion	Beispiel
`sll`	*Shift Left Logical*	sll rd, rs1, rs2	rd ← rs1 << rs2	sll t0, t1, t2
`slli`	*Shift Left Logical Immediate*	slli rd, rs1, u	rd ← rs1 << u	slli t0, t1, 2
`srl`	*Shift Right Logical*	srl rd, rs1, rs2	rd ← rs1 >> rs2	srl t0, t1, t2
`srli`	*Shift Right Logical Immediate*	srli rd, rs1, u	rd ← rs1 >> u	srli t0, t1, 2
`sra`	*Shift Right Arithmetical*	sra rd, rs1, rs2	rd ← rs1 >> rs2	sra t0, t1, t2
`srai`	*Shift Right Arithmetical Immediate*	srai rd, rs1, u	rd ← rs1 >> u	srai t0, t1, 3

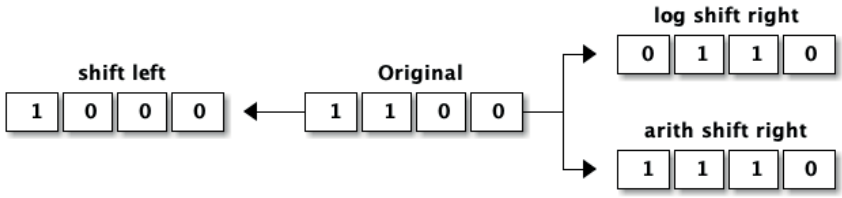
Dabei tun sich zwei Fragen auf: Was passiert mit den Bit, die aus dem Register geschoben werden? Und was kommt an die freigewordenen Stellen auf der anderen Seite?

Zu den herausgeschobenen Bit haben wir gute Nachrichten: Sie werden auf einen wunderschönen Bauernhof gebracht, wo sie ihre Tage mit lustigen Spielen verbringen, umgeben von Freunden. Dass wir nie wieder etwas von ihnen hören, liegt daran, dass sie zu glücklich sind, um sich zu melden.

Bei den Nachrückern unterscheiden wir zwei Varianten. Bei der ersten, »logischen« Verschiebung werden die freien Stellen mit Nullen aufgefüllt. Bei der zweiten Variante, der »arithmetischen«, unterscheiden sich Verschiebungen nach links und rechts. Wenn es nach *links* geht, wird ebenfalls eine Null eingefügt, womit kein Unterschied zur logischen Verschiebung besteht. Entsprechend bieten viele Prozessoren – auch RISC-V – keine gesonderten Befehle für arithmetische Linksverschiebungen an. Bei einer arithmetischen Verschiebung nach *rechts* wird dagegen das MSB in die freie Stelle kopiert, damit das Vorzeichen erhalten bleibt.

Bei den Nicht-Immediate-Formen wie `sll` werden bei RV32 nur die untersten *fünf* Bit in `rs2` berücksichtigt, bei RV64 sind es die untersten *sechs*. Die Varianten wie `slli` begrenzen den Immediate-Wert entsprechend. Damit kann eine Verschiebung bei RV32 um 31 Stellen erfolgen und bei RV64 um 63 Stellen. Negative

Werte sind nicht erlaubt (daher `u` und nicht `i` in der Tabelle). Getrennte Rotationsbefehle (auch »zyklische Verschiebungen« genannt) gibt es bei RISC-V nicht.



Verschiebungen werden gerne bei Assembler benutzt, um mit Zweierpotenzen zu multiplizieren. Das geht auch bei Prozessoren, die keine Hardware für die Multiplikation mitbringen.

Befehl	Entspricht	Faktor
slli, t0, t0, 1	2^1	2
slli, t0, t0, 2	2^2	4
slli, t0, t0, 3	2^3	8
slli, t0, t0, 4	2^4	16
slli, t0, t0, 5	2^5	32
slli, t0, t0, 6	2^6	64

Mit Kombinationen aus Verschiebungen und einzelnen Additionen können wir dann auch andere Multiplikationsaufgaben lösen, etwa indem wir statt `x*5` in Assembler `(x<<2)+x` rechnen.

Wir können mit einem Trick außerdem eine Vorzeichenerweiterung erzwingen. Nehmen wir an, bei einem RV64-Prozessor haben wir eine eigentlich negative 60-Bit-Zahl wie 0x0FFF_FFFF_FFFF_FFFF, deren Vorzeichen für die weitere Bearbeitung auf 64 Bit erweitert werden muss. Wir verschieben sie zuerst vier Stellen nach links bis an den »linken Rand« des Registers. Dann schieben wir sie arithmetisch um dieselbe Zahl von Stellen wieder zurück.

```
li t0, 0x0FFFFFFFFFFFFFFF     # note first nibble is zero
slli t0, t0, 4
srai t0, t0, 4
```

In `t0` liegt anschließend 0xFFFF_FFFF_FFFF_FFFF.

10.3 Sprünge und Verzweigungen

Richtige Computer brauchen Schleifen und Verzweigungen und damit Entscheidungen und Sprünge. Wir sprechen von *control flow instructions*, weil sie den »Fluss« des Programms ändern. Allen Sprüngen und Verzweigungen ist gemein-